LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 824-829
cop. 2

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

OCT 19 1977

L161—O-1096

UIUCDCS-R-76-828

VALIDATION OF THE ANALYSIS PASS OF THE CLEOPATRA COMPILER

by

SANDRA ANN LEACH

October 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the
JAN 20 1977
University of Illinois
at Urbana-Champaign

Digitized by the Internet Archive
in 2013

http://archive.org/details/validationofanal828leac

VALIDATION OF THE ANALYSIS PASS OF THE CLEOPATRA COMPILER

BY

SANDRA ANN LEACH

B.S., University of Illinois, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Urbana, Illinois

Acknowledgments

Preparing a thesis is not only a technically demanding process, but a lonely and frustrating one as well. I have been fortunate to have many colleagues and friends who have supported me in both respects.

On the technical side, I would first like to thank my thesis advisor, Dr. M. George Friedman, Jr., for suggesting this project, for his helpful comments while reading this thesis, and for his patience. I also wish to thank Dr. Axel T. Schreiner for his advice in the planning of the project, and Mr. Scott Fisher for his cooperation and assistance.

My heartfelt appreciation and thanks go to my good friends, Dr. R. L. Danielson, Mr. W. D. Gillett, Ms. M. J. Irwin, Dr. A. C. Weaver and Dr. T. R. Wilcox, for their help in editing and proofreading this thesis, and most of all for their moral support.

Last and most importantly, I thank my parents for their unflagging confidence and encouragement. Without them this thesis would not have been possible.

Thank you all.

S.A.L.

Table of Contents

1 Introduction 1

2 Testing Philosophy 4

2.1 Definition of Validation 4

2.2 Available Testing Methods 5

2.3 Selecting a Method 8

2.4 Implementation of the Testing 10

3 Lexical Analysis 11

3.1 Character Set Mapping 11

3.2 Delimiting Boundaries 13

3.3 Comment Tokens 15

3.4 Identifier and Operator Tokens 16

3.5 Numeric Constants 18

3.6 Character Literals 23

3.7 Complete Lexical Analysis 25

4 Syntactic Analysis 30

4.1 Structure and Data Blocks 31

4.2 Routine Blocks 37

4.3 Expressions 42

5 Semantic Analysis 50

5.1 Symbol Table Complex 50

5.1.1 Predefined Symbols 50

5.1.2 User-defined Symbols 52

5.2 Intermediate Text 53

6 Conclusions 55

References 57

Appendix: Summary of Productions Implemented 59

1 Introduction

CLEOPATRA, the "Comprehensive Language For Elegant Operating System And Translator Design", was designed in 1974 by Schreiner [1,2] as a high-level, block-structured, extensible language aimed primarily at operating systems implementors.

During 1975 a first attempt at an implementation of the language was made in PL/I on an IBM/360. This version implemented only a subset of CLEOPATRA and was intended to be a bootstrap so that future CLEOPATRA compilers could be written in the language itself. The compiler consisted of two phases: analysis and code generation. The former performed lexical, syntactic, and semantic analysis and produced a symbol table and intermediate text. The latter phase operated on the intermediate text to produce object code in the form of a relocatable binary deck suitable for input into the OS/360 loader.

Unfortunately, soon after the completion of this project the analysis phase implementation was found to be inadequate; it was bug-ridden and, because of inferior coding, would have required substantial effort to correct. Therefore, the decision was made not to attempt repair of the bad code, but to rewrite it. During 1976 a second attempt was made; the implementation of the old code generation phase (Halbur [3,4]) was considered to be of sufficient quality to retain, at least for the present, so the new analysis phase (Fisher [5]) was written to fit the coder's specifications.

The faults of the original analysis phase and their detection at such a late date resulted from meager and ad hoc testing methods during its development. In order to preclude running amok in the same fashion a second time, an extensive testing effort has been made concurrently with the development of Fisher's implementation; a documentation of the testing effort comprises the body of this thesis.

The goal of the testing is twofold: first, to demonstrate that Fisher's implementation of the analysis phase is "correct" (or, more realistically, to document any deviations from the specifications) so that later CLEOPATRA compilers can be assured of a solid basis; and second, to produce a test data package which can be permanently associated with the compiler and used for regression testing when the currently implemented subset is expanded.

Chapter 2 discusses the difficult problem of testing a large piece of software and the basic philosophies that this testing effort follows. Subsequent chapters detail the design, production, and execution of test cases for each component of the compiler's analysis phase: Chapter 3 on lexical analysis, Chapter 4 on syntactic analysis, and Chapter 5 on semantic analysis. The final chapter summarizes the results. Appendix A gives the definition of the language actually accepted by this implementation of the analysis phase, and Appendix B presents the complete test data generated.

This thesis is intended as a companion to the reports mentioned above. A knowlege of the information contained in [1] and [4] is essential to understanding the material discussed here. To minimize confusion, the terminology and notation used in this thesis are, as much as possible, consistent with the earlier reports.

2 Testing Philosophy

The intent to validate a piece of software immediately gives rise to many questions. Some of them are:

- What does validation mean?
- What testing methods are available?
- Which method is to be used and why?
- How is the method to be carried out?

Before discussing the actual validation of CLEOPATRA's analysis phase, it would be beneficial to answer these questions.

2.1 Definition of Validation

Gruenberger [6] gives a very useful definition of validation. Assume that a program has been designed and encoded, and the implementor has debugged the code, thus removing the mechanical errors of coding and producing a superficially correct program. The program as written clearly solves some problem; the purpose of validation is to show that it solves the desired problem.

Validation usually involves testing performed by an outside agent playing the role of the "Devil's Advocate". This testing has a dual purpose: showing that all functions in the program specifications are implemented, and showing that all implemented functions are specified. These are what Elmendorf [7] calls "specification testing" and "program testing," respectively. If the two sets of functions are identical, the correct problem has been solved, and the program has been validated.

However, if the two sets of functions are not identical, the testing will have pinpointed the aberrant function(s). The cause of the deviation could be inconsistent, poorly-stated or erroneous specifications or miscoding (or even a combination of these). The implementor must then decide how best to remedy the discrepancy and carry out the correction; the testing can then be repeated.

The repetition of the entire bank of test cases run against the software after each repair of the code is called "regression testing" (Brown and Sampson [8]). Any modification of the program runs the risk of introducing an error which was not present before the change was made and which may be totally unrelated to the successful implementation of the change itself. So testing must be repeated to insure that previous work has not been damaged in some obscure way by the alteration.

2.2 Available Testing Methods

There have been many methods of software testing discussed in the literature during the past few years. Some of these techniques are more thorough than others, both in a formal sense (Goodenough and Gerhart [9]) and in relation to the concepts of program and specification testing as discussed above. Some are more costly in human time or computing resources or both. Therefore a practical balance between thoroughness and expense must be found.

One way to show the correctness of a program is by proof. However, there is rarely enough information available to allow a formal proof of correctness. As London [10] points out, assumptions must be made about the semantics of the programming language used, the completeness of the problem domain specification, and the correctness of the proof itself. Going a step further, the entire running environment of the program (language, operating system, and hardware processors) must be completely axiomatized and proved consistent before the program can be formally implemented and proved within the system. This is clearly beyond current state-of-the-art techniques, so more informal proof methods are employed. But these less formal proofs are more prone to error. Goodenough and Gerhart show that several programs which have been "proven" in the literature contain numerous bugs. It is interesting to note that many of these bugs are due to imprecisely stated specifications and would have been found if even rudimentary test cases had been run.

Another testing method, which relies on brute-force computing power, is to test all possible input combinations. Even for a program having a very small number of inputs, exhausting input combinations is absurdly impractical. Take, for example, a program with two inputs X and Y that computes one output Z=F(X,Y) (Huang [11]). If X and Y are represented in 32-bit registers, there are $2^{32}*2^{32}=2^{64}$ possible input combinations. Even if the program itself took an average of only one millisecond per execution, it would take more than 50 billion years to complete the test!

The next three methods involve similar approaches. The first method is to generate test cases that exercise every statement of the program. This test is practical, since choosing the test data is a fairly straight-forward, mechanical process. However, there are several types of errors that are not necessarily discovered using this technique. Erroneous transfer of control may not be found since it is possible to execute every statement in a program and not traverse all (possibly faulty) control paths. Also, missing control paths (i.e., failure to examine a special case) probably will not be discovered since the choice of test data is based on the program code and not on the program specification.

The second method, which is much more likely to detect errors, is to require that each and every control path be executed at least once. This method guarantees that erroneous transfers of control will be found, and also detects subtle errors which are contingent on the exact sequence of previous events in the execution. However, the method is impractical due to loop constructs. A program with a loop has at least as many different control paths as the number of times the loop can be iterated, and for programs with several loops the total number of paths is the product of the paths through each individual loop. Obviously, this can lead to a prohibitively large number of paths in many cases.

A more practical approach, a compromise between the previous two, is to generate test data that exercises all program statements and all branches. This is equivalent to requiring

that each edge of the flowchart corresponding to the program be traversed at least once (Huang [11]). This scheme is reasonably effective, but still does not insure that all errors will be detected; those such as missing control paths, incorrect path selection, and an incorrect or missing action may not be found due to an (un)fortunate choice of test data.

The last technique to be discussed has a completely different emphasis. Instead of basing tests on the program itself, this method generates test cases directly from the specifications, using program structure only to complement the specification information. Goodenough and Gerhart describe how decision tables may be employed to facilitate identification of "all conditions relevant to a program's correct operation," from which test cases are generated that exercise all possible combinations of these conditions. This is an effective technique, but it involves an element of insight on the part of the tester to correctly determine the "relevant conditions" from the specifications and program.

2.3 Selecting a Method

Clearly, some of the techniques described above are more suited than others to a project such as verifying a compiler . The criteria used in rating these methods include practicality, effectiveness, and expense.

Attempting to formally prove a piece of software as substantial as a compiler is an exercise in complexity and frustration. The applicable techniques are extremely tedious and

rapidly become unmanageable. Use of less formal techniques leads to "proofs" which are much less conclusive than one would desire. Testing by exhausting input combinations is similarly impractical; the complexity of the input domain of the CLEOPATRA compiler results in sets of test data which are intolerably large. Random sampling is feasible, but it is very difficult to objectively measure the degree of success of this method since the test process is entirely subjective. Using test data that exercise all statements of a program is easier to evaluate, but is not effective enough in exposing errors. Exercising all program paths is impractical due to the sheer number of possible paths; the problems in using this method are similar to those encountered with exhaustive input testing.

Two testing attacks remain: exercising all program statements and branches, and exercising all combinations of specification conditions. The former approach implements program testing, but ignores the specifications; the latter implements specification testing, but tests the program only to a lesser degree. Since neither of these individually constitute a sufficient test, the logical answer is to combine the two approaches, obtaining a testing method that performs both program and specification testing. Therefore, the attack to be taken in the validation of the CLEOPATRA analysis phase is to first generate test cases from the specifications and from a limited knowledge of the program's structure so that all combinations of relevant conditions are exercised, and then to verify that these test cases do indeed cause the execution of all program statements and branches.

2.4 Implementation of the Testing

Whereas good program design is performed top-down, program testing is normally performed bottom-up. Hetzel [12] advocates "segmentation," the concept of testing new functions only after testing all sub-functions. The advantages of segmentation are increased conceptual clarity, economy of test cases, and the ability to do more testing in parallel with program development.

This modular approach was used in the validation effort to be discussed. Test cases for sub-functions were generated using Goodenough and Gerhart's decision table technique, then the test cases for all the sub-functions were synthesized to test the complete function. This last, unified test was run on the PL/I Optimizing compiler, which offered an option of counting statements executed and branches taken, in order to verify that all program statements and branches were exercised.

The major modules considered in this validation were lexical, syntactic and semantic analysis. Each of these modules was tested bottom-up, beginning with lexical analysis, and the correctness of each successive module depended on the correctness of the preceding module. The bulk of this thesis is composed of a detailed account of the validation of each of these major modules. The specifications used for the testing were the original language definition (Schreiner [1], hereafter called "the Report"), abbreviated by Halbar's subset definition [4], and modified somewhat by Fisher's implementation considerations [5].

3 Lexical Analysis

Because lexical analysis, the identifying and isolating of tokens from an input stream, is the foundation of any compiler, the testing of this module was very thorough. The aim was to insure that any legal token (and many marginally illegal ones) would be correctly isolated from any possible environment. This resulted in a greater concern with how the lexical analysis handled illegal input than with how later modules processed illegal input. The sections of this chapter describe the sub-functions that were tested, including the character set mapping, the detection of delimiting boundaries, and the isolation of tokens (comments, identifiers, operators and constants). The last section discusses the integration of all sub-function test cases and the testing of the complete lexical analysis module.

3.1 Character Set Mapping

The kernel of lexical analysis is the mapping of the input characters into the functional classes of character types (letter, digit, delimiter, special and control). The specifications against which the code was tested were productions 1.1 through 1.5 of the Report. The code for this mapping function was isolated, and all characters that could be input via a DEC/10 terminal were presented to the function. The results were as follows. (See Appendix A for an explantion of the BNF notation.)

(1.1) letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

(1.2) delimiting_character ::= (|) | . | : | , | ' | ; | blank_character | _

(1.3) special_character ::= @ | # | $ | % | & | * | - | + | = | " | ? | / | < | > | ⊥ | ¢ | ¬

(1.4) digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(1.5) control ::= !

Productions 1.1 and 1.4 agree exactly with the specifications. Production 1.2 differs only in that the underscore character has been added; this will be discussed further with identifiers. The characters ≤ and ≥ do not appear in production 1.3, having been deleted from the character set because of the inability to enter them via input devices available at this installation. Square brackets [and] have also been deleted from the character set. The user is cautioned that the character ¢ (which is entered as a ¢ on a keypunch, but as a [on a DEC/10 terminal) prints on the PN train printer as a [; this is due to the internal character mapping of the system link between the DEC/10 and IBM/360, and a policy decision was made to let it remain so. Production 1.5 differs from the specifications in that the backspace character was deleted, again because of linkage problems, and the end_of_source_record "character" is not mentioned here since it is more a state of the lexical analyzer and is therefore not given a mapping.

3.2 Delimiting Boundaries

The delimiting boundaries (as described in section 1.4 of the Report) between the characters of the five character classes are defined by the following rule: there is an understood delimiting boundary between any two characters, except that there is no delimiting boundary between

- two letters
- a letter and a digit
- a digit and a letter
- two digits
- two special characters.

In order to verify that this rule holds, a decision matrix was constructed representing the cross product of the set of five character types with itself. That is, an entry in row i and column j of the matrix indicates that a character of type j immediately follows a character of type i. Test data were generated so that there was a boundary corresponding to each position in the matrix. Figure 1 shows the test data along with the matrix, where each position is filled with the character pair satisfying that boundary.

The results of the test, in the form of tokens identified, are listed in Figure 2. These results agree with the boundary rule except in the case of a digit followed by a letter: if the digit (or digit string) is immediately preceded by an alphabetic character, there is no boundary as the rule predicts; however, if the digit string is immediately preceded by a non-alphabetic character, then there is a delimiting boundary between the last

	letter	digit	delim	special	control
letter	AB	B7	R;	M=	L■
digit	3R	83	7:	6*	9■
delim	,T	:8	.,	;@	)■
special	<G	$6	=.	@$	*!
control	■A	■4	■(	■<	■!

■ indicates end_of_source_record

```
AB7:83R;@$6*
4                                                                    L
(                                                                    9
<G5E                                                                 )
!
M=.,T
```

Figure 1
Delimiter Matrix and Test Data

digit of the string and the following letter. This rather confusing exception is illustrated by two examples from the test data: in the sequence 'AB7:83R' there <u>is</u> a boundary between the '3' and the 'R' since the digit string '83' is preceded by a ':'; on the other hand, in the sequence '<G5E' there is <u>no</u> boundary between the '5' and the 'E' since the digit string '5' is preceded by a 'G'.

This minor deviation from the specifications was not corrected since it would have involved a fairly major redesign of the scanner. As will be seen later, this discrepancy, however small it might seem, causes adverse side-effects in the processing of numeric constants that contain illegal characters.

```
card #1:  AB7  :    83   R    ;    @$   6    *
card #2:  4    L
card #3:  (    9
card #4:  <    G5E  )
card #5:  M    =    .    ,    T
```

Figure 2
Tokens Identified from Delimiter Test

3.3 Comment Tokens

Comments are specified in section 1.5 of the Report as:

(1.6) comment ::= COMMENT any sequence of characters with the exception of a semicolon ;

(1.7) ::= ! any sequence of characters with the exception of an end of the source record

end_of_source_record

The only change in Schreiner's specifications is that a comment is not equivalent to a blank character, as stated in the Report; it does act as a general delimiter where it appears, but it may not occur wherever a blank character is legal. In particular, a comment is not allowed within a constant (as described in section 3.5) though blanks are allowed.

A reasonable test case for comments is to construct a comment of each type containing every possible character (including an undefined character, but of course no semicolon in the first type of comment), along with the string 'COMMENT'. This insures that neither single characters nor "embedded comments" interfere with the isolating of the comments. Figure 3 shows the test cases executed. (Notice that the string 'COMMENT'

must be surrounded by delimiting boundaries.) The result was that the tokens 1, 2, 3 and 4 were correctly isolated, and the comments were properly ignored.

```
|1COMMENT 0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ !().:',_@#$%|&*-+="<>?/[ ]COMMENT |
|abcdefghijklmnopqrstuvwxyz ; 2     !COMMENT abcdefghijklmnopqrstuvwxyz           |
|3!ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 !@#$%|&*()_+-=][" ':;<>?,./             |
|4                                                                               |
```

Figure 3
Test Cases for Comment Tokens

3.4 Identifier and Operator Tokens

The main specifications for identifier and operator tokens are again taken from the Report (section 2.1):

(2.1) identifier ::= letter [letter | digit]•

(2.2) operator ::= { special_character }• | identifier

There are only two exceptions to the discussion in the Report. First, the transparent underscore is not implemented; an underscore appearing within an identifier acts as a delimiter, thus dividing the identifier. (This is the reason the underscore was added to the character set as a delimiting character.) Second, there is not explicit maximum length on either identifiers or operators; they are limited in length only by the restriction that they cannot be broken across source record boundaries. Therefore, in this implementation these tokens may be up to 80 characters long, if positioned properly.

From these specifications the decision tables in Figure 4 were produced, enumerating cases to be tested for identifiers and operators. Figure 5 shows actual test cases satisfying the criteria of the decision tables and the results after execution. All tokens were isolated correctly, verifying the correctness of the identifier/operator handling portion of the lexical analysis.

Identifiers

legal cases:	1	2	3	4	5	6	7	8
ends with letter or digit	L	L	L	L	D	D	D	D
contains interior letters	N	Y	N	Y	N	Y	N	Y
contains interior digits	N	N	Y	Y	N	N	Y	Y

Pathological Cases: starts with a digit
broken across card boundary
contains a blank
contains an underscore
contains other illegal character

Operators

legal cases:	1	2
length>1	Y	N

Pathological Cases: broken across card boundary
contains a blank
contains other illegal character

Figure 4
Decision Tables for Identifiers And Operators

```
I  D0918273645D A9BCD8EFG7HIJ6KLM5NOP4QRS3TUV2WXYZ10 ZYXWVUTSRQPONMLKJIHGFEDCBA
NUMBERS  UNDER_SCORE  ABCD0EF1GH2IJ3KL4MN5OP6QR7ST8UV9WXYZ  G0123456789  K9    =
@#$%&*-+="?/<>|[¬ $+->:<*/$   7WRONG  A*TIMES  RIGHT5  SEMI;COLON
  numbers     IDENTifier
```

Figure 5
Identifier and Operator Test Cases

3.5 Numeric Constants

The specifications for constants, as are the specifications for the remainder of this testing, are based on Halbur [4] with only minor modifications. Halbur's productions 2.1 through 2.14 are correct, but 2.15 and 2.17 have been changed to:

(2.15) long_integer ::= F. decimal_string

(2.17) bit ::= S. binary_digit

This change in the BNF reflects the restriction that only one of a base or a size can be specified in one constant (e.g., F.X.7FFF is not legal, even though X.7FFF and F.32767 are legal). The only other modification is that one or more blanks are allowed between the letter-dot preface and the minus symbol, if it appears, or between the preface and the digit string if there is no minus present.

The decision table given in Figure 6 was constructed from the specification productions, and the test cases shown in Figure 7 were generated to satisfy the decision table conditions. The results were not completely as expected.

The major problem hinged on constants that exceeded the specification bounds for minimum and maximum values (see Figure 8). First, no provision had been made in the code to detect

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
PREFIX[1]	N	N	N	N	N	N	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	F	F	F	F	F	F	F	F	F	F	F	F
BLANKS AFTER .	(N)	(N)	(N)	(N)	(N)	(N)	N	N	N	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	N	N	N	Y	Y	Y
CONTAINS MINUS	N	N	N	Y	Y	Y	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
BLANKS AFTER -	(N)	(N)	(N)	N	N	Y	(N)	(N)	(N)	(N)	(N)	(N)	N	N	N	N	N	N	Y	Y	Y	(N)	(N)	(N)	(N)	(N)	(N)	N	N	N	N	N	N
KIND OF DIGITS[2]	D	I	D	D	D	D	H	H	I	H	H	I	H	H	I	H	H	I	H	H	I	D	D	I	D	D	I	D	D	I	D	D	I
OUT OF BOUNDS	N	-	Y	N	Y	N	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-
LEGAL	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(N)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)

	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66
PREFIX[1]	F	F	F	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
BLANKS AFTER .	Y	Y	Y	N	N	N	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	Y
CONTAINS MINUS	Y	Y	Y	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
BLANKS AFTER -	Y	Y	Y	(N)	(N)	(N)	(N)	(N)	(N)	N	N	N	N	N	N	Y	Y	Y	(N)	(N)	(N)	(N)	(N)	(N)	N	N	N	N	N	N	Y	Y	Y
KIND OF DIGITS[2]	D	D	I	B	B	I	B	B	I	B	B	I	B	B	I	B	B	I	B	B	I	B	B	I	B	B	I	B	B	I	B	B	I
OUT OF BOUNDS	N	N	-	N	Y	-	N	Y	-	N	N	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-	N	Y	-
LEGAL	(N)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(N)	(N)	(N)	(Y)	(N)	(N)	(Y)	(N)	(N)	(N)	(N)	(N)	(N)	(N)	(N)	(N)	(N)	(N)

[1] X., B., F., S., NONE

[2] Binary, Decimal, Hex, Illegal

\- = doesn't matter
Y = yes
N = no
(Y) = necessarily yes
(N) = necessarily no

Figure 6
Decision Table for Constant Specifications

```
|32767  458   10   57B3  601298   -32768  -1592  -69437  - 2914
|X.7FFF  X.5F98  X.8FF1   X.4G7   X. 0CDE   X.  FFF3   X. 4LA9   X.-8000
| X.-ABCD   X.-5Q31   X. -23EF   X. -F93A   X.  -M12   X. - 73B
|  X. - 9ABE   X. - 35D   X . 456E
| F.2147483647   F.5147483609  F.1234567ABC F. 1000000000  F. 9876543210
|   F. 215Z137  F.-2147483648   F.-27345618900  F.-983A76   F. -2147483648
|F. -3987654021    F.  -165291C257  F.  - 2147483648  F. - 39875642301
|F. - 529816D2C
|B.111111111111111  B.1000000100100000 B.10141011  B.  111111111111111
| B. 10000000000000000  B. 00110F011   B.-10000000000000000
|B.-11000111000111000  B.-1010Q0110  B.  -1110001   B. -10111000111000111
|  B. -00111610   B. - 01000101  B. - 11000111101010011  B. - 001012110
|B.- 1100   X.- 56FE   F.- 34345654
| S.01  S.1  S.1101 S.2  S. 1  S. 010  S.012   S.-1   S.-101  S.-1A0
|S. -0  S. -0111  S. -1C0   S. - 1   S. - 10   S. - D01
```

Figure 7
Test Data for Constants

constants whose values were out-of-bounds. Therefore, the first time an extremely large constant was encountered, the FIXEDOVERFLOW condition occurred and aborted the execution of lexical analysis. Clearly, this drastic an error "message" was undesirable, so an ON FIXEDOVERFLOW unit, which printed a reasonable error message and set the constant's value to zero, was incorporated into the procedure that calculates constant values, and the test was rerun. Unfortunately, this repair did not completely correct the problem. This time constants of extremely large magnitude (>2147483648) were correctly detected, but decimal constants whose values were only moderately out-of-bounds (32767<|x|<2147483648) were not flagged, and both decimal and long_integer constants in this range were given spurious values. The cause of this turned out to be the mixing of FIXED BINARY 15 and 31 variables to hold the constant's computed value, causing the loss of high-order significant bits. The problem was finally solved by uniformly using FIXED BINARY 31 variables to hold the computed value and checking normal decimal constants to

insure that their values were within the correct range.

Second, the value F.-2147483648 was flagged as being out of bounds. The reason for this was that, by definition, the value of a negative constant is -1 times the positive value computed. Therefore, when the positive value 2147483648 was computed, it was correctly flagged as out-of-range before the value could be negated. This indicates that the specifications were inconsistent; the range of long_integer constants should be |long_integer| < 2147483648, and not as shown in Figure 8.

	MAXIMUM	MINIMUM
DECIMAL	32767	-32768
HEXADECIMAL	X.7FFF	X.-8000
LONG_INTEGER	F.2147483647	F.-2147483648
BIT	S.1	S.0

Figure 8
Range of Constant Values

A third problem that this test data revealed was that constants with an F. preface which contained any letters A through F were evaluated interpreting these letters as hexadecimal digits. The cause of this problem was a local character set mapping for digits that distinguished between binary, octal and other digits, but did not distinguish between decimal and hexadecimal digits. The mapping was modified, and a rerun of the test data showed that the problem was solved.

The fourth inconsistency involves the processing of those constants containing illegal blanks following the minus sign. When a negative decimal constant contains blanks separating the minus sign and the digit string, the string is interpreted as an operator followed by a positive decimal constant, which is correct according to the specifications. If a hexadecimal or binary constant contains one or more blanks following the minus, the entire token is correctly isolated and flagged as being in error, which again is what the specifications stipulate. However, if a long_integer constant contains illegal blanks after the minus, the F.- is tokenized and given a value of zero; this forces the string that would have been the body of the constant to be analyzed on its own individual characteristics (being typed as either a decimal string or identifier or decimal string followed by an identifier) giving unexpected and unwanted results. Similarly, if an S. is followed by any character other than a binary digit or a string of blanks followed by a binary digit, the S. is tokenized and given a value of FALSE, and the rest of the string is interpreted on its own. The processing of long_integer constants was changed so that it was consistent with hexadecimal and binary constants (i.e., the entire token was isolated and flagged as illegal), but it was decided to let the bit constant handling remain as it was.

The last problem uncovered by this test data was the handling of those numeric constants containing illegal characters, such as X.-5Q31 or 57B3. In cases like these, the constant is interpreted as containing all the characters upto but not including the illegal character; then everything from that

character on is interpreted on its own. As a result, these constant are interpreted as a legal constant followed by another token (or tokens), and not as a single illegal constant as one would expect, since it is in general more probable that the illegitimate letter was a typographical error (as in using the letter O instead of the digit zero) or resulted in using a digit of an incorrect base. The solution to this problem mainly consists of repairing the digit/letter boundary problem discussed in section 3.2, which, as stated before, was not done.

3.6 Character Literals

Character constants are defined by:

(2.28) literal_value ::= C. any sequence of characters up to but not including the first following blank_character

That is, the first character following the C. preface begins the literal_value, and the first blank_character following the C. ends the literal_value. Additionally,

- End_of_source_record is transparent to a literal_value and does not become a part of it.
- The underscore character represents a blank_character in the literal_value.
- Neither a ! nor the string COMMENT initiate comments; they are considered part of the literal_value.
- A literal_value may not exceed 256 characters; longer literals are truncated to 256.

Figure 9 shows the literal_value test cases. Contained in at least one of the literals are ! and COMMENT, each character of the character set, an undefined character, an underline, and an end_of_source_record (but no blank_character, of course). The data also include a literal longer than 256 characters and one of length zero.

```
C.ABCDEFGHIJKLMNOPQRSTUVWXYZ_().:;,'_@#$%&*-+/="¬><|?_0123456789_!THISISNOTACOMM
ENTBUTALONGCHARACTERSTRINGCONTAININGEVERYCHARACTERINTHECHARSET_ITISVERYLONGBUTST
ILL<256!!abcdefghjijklmnopqrstuvwxyz              C._      C.
C.ABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789_@#$%&*-+="/><¬|?_().,;:'_!THISLITERALALS
OCONTAINSEVERYCHARACTERINTHECHARACTERSET_COMMENT_BUTITISMUCHLONGERTHAN256ANDWILL
BETRUNCATEDSOTHATITISLESSTHANTWOHUNDREDFIFTYSIXCHARS;HOWEVERITGETSINCREASINGLY#?
DIFFICULTTOWRITECHARACTERSTRINGSTHATARESOVERYLONG[abcdefghijklmnopqrstuvwxyz?]¢
                          C.THISISJUSTASHORTLITERALTHATHAPPENSTOENDINCOLUMN80
```

Figure 9
Character Literal Test Cases

The first execution of this test was literally a disaster. Literals that were contained completely on one line and surrounded by blanks were handled correctly, but strings containing end_of_source_records were badly mangled. The first occurring end_of_source_record ended the literal, and, if 'x' was the string between the C. and the first end_of_source_record, the token "recognized" became 'xx' (i.e., the string concatenated with itself). The remainder of the literal was interpreted as a sequence of new tokens.

The main cause of these problems was the procedure that read in a new line of source text and the conditions this procedure expected to be true when invoked; the procedure was designed to do more processing than it logically should have, and by doing so

thwarted the handling of character literals. These problems were repaired, and the tests were rerun. This time all literal_values were correctly isolated.

3.7 Complete Lexical Analysis

After having tested each portion of the lexical analysis individually, it was necessary to integrate all these tests together in order to test the entire lexical analysis phase. The motivation behind the design of the composite test was the following: assume all tokens in the input stream have been correctly processed (i.e., as predicted) up to a given point; the correct isolation of the next token now depends only on what follows it. (The fact that the lexical analyzer is a procedure that is called each time a token is needed for the parse was part of the basis for making this assumption.)

The first step in the design of this test was to generate a list of specific token types that had been previously tested individually. Then this list was expanded by differentiating, within each token type, tokens that might have distinct ending characteristics (e.g., identifiers ending in a letter vs. identifiers ending in a digit). The original list was again expanded by differentiating on beginning characteristics (e.g., decimal constants beginning with a decimal digit vs. those beginning with a minus sign). Figure 10 shows both complete lists. Character literals were not included in the distinct endings group since these must all be followed by a blank_character which is not considered a part of the literal.

DISTINCT ENDINGS	DISTINCT BEGINNINGS
1 delimiter character	1 delimiter character
2 comment: COMMENT ;	2 comment: COMMENT ;
3 comment: ! end_of_source_record	3 comment: ! end_of_source_record
4 identifier ending in a letter	4 identifier
5 identifier ending in a digit	
6 operator	5 operator
7 decimal constant	6 positive decimal constant
	7 negative decimal constant
8 hex constant ending in a letter	8 hex constant
9 hex constant ending in a digit	
10 long_integer constant	9 long_integer constant
11 binary constant	10 binary constant
12 bit constant	11 bit constant
	12 character literal
13 end_of_source_record	13 end_of_source_record

Figure 10
Token Classes Used In Composite Test

A matrix was constructed, similar to that used in the delimiter test in section 3.2, where the rows were numbered with the classes of tokens possessing distinct ending characteristics, and the columns numbered with the classes of tokens having distinct beginning characteristics. (See Figure 11.) The idea here is to insure that each token with a distinct ending is followed in the test case by each token with a distinct

beginning. Another important criterion of this composite test is that it include every instance from the sub-function tests, in order to preserve the "testedness" of the sub-functions.

The actual test cases were then generated, filling in the matrix as follows: the entry in row i and column j of the matrix received the value 'n/m' when in the test data a token of type j followed a token of type i on card n, and the first character of the type j token appeared in card-column m. For example, the entry in row 4 and column 7 is 3/11; this indicates that an identifier ending in a letter (IDentifier) immediately precedes a negative decimal constant (-65000), where the minus sign is the 11th character on line 3 of the test data. The actual test data appear in Figure 12. It is easy to verify that all sub-function tests are included in this data.

The results of this test reaffirm that previous repairs had been successful, but brought to light one more problem. If any numeric constant was followed immediately by a negative decimal constant (or an operator beginning with a "-"), the two were recognized together as a "constant" (i.e., B.0110-35); when evaluation was then attempted, an error occurred because the interior minus was not a digit. Rerunning the test showed that the first repair made for this problem was incorrectly performed, which wreaked havoc everywhere. The repairs themselves were then fixed, and a re-execution of the data showed that indeed the problem was finally solved.

DISTINCT BEGINNINGS

DISTINCT ENDINGS	1	2	3	4	5	6	7	8	9	10	11	12	13
1	3/33	3/34	9/56	3/56	1/9	1/3	2/54	11/2	7/29	5/45	11/17	12/6	12/81
2	12/80	12/65	6/27	3/49	2/47	14/30	19/18	1/56	10/57	15/65	17/41	5/24	18/81
3	14/1	5/1	8/1	9/1	18/1	20/1	7/1	17/1	22/1	21/1	10/1	23/1	***
4	1/2	* 15/12	17/70	* 11/14	1/74	* 4/22	3/11	* 13/20	* 18/52	* 19/29	* 20/33	* 21/18	3/81
5	3/55	* 27/19	4/24	* 13/17	3/60	* 13/16	9/3	* 20/8	* 20/20	* 19/36	* 21/27	* 21/37	2/81
6	13/26	1/12	19/42	4/17	* 10/6	3/18	* 10/8	3/62	4/2	11/45	1/76	2/50	10/81
7	1/8	10/28	16/36	1/68	3/17	* 7/5	10/23	10/11	3/20	2/60	7/7	9/8	15/81
8	17/10	* 18/7	21/65	** 21/49	11/43	* 17/4	11/18	7/16	* 22/18	* 10/16	3/69	* 22/34	11/81
9	12/5	* 18/22	22/69	** 17/66	10/77	* 22/53	1/63	17/5	* 23/24	* 23/37	7/24	* 24/9	24/81
10	3/32	18/40	7/42	13/13	4/16	* 17/54	14/41	10/71	17/57	23/50	16/23	24/27	25/81
11	15/10	5/55	23/66	2/77	15/27	** 16/5	15/76	11/64	16/7	12/31	12/37	14/55	26/81
12	7/28	12/41	20/42	3/76	10/5	** 24/42	16/33	7/10	17/46	11/19	16/29	24/50	1/81
13	11/1	2/1	28/1	3/1	4/1	25/1	26/1	12/1	13/1	16/1	19/1	27/1	***

* illegal ** sometimes illegal *** physically impossible

Figure 11
Decision Matrix for Composite Lexical Analysis Test

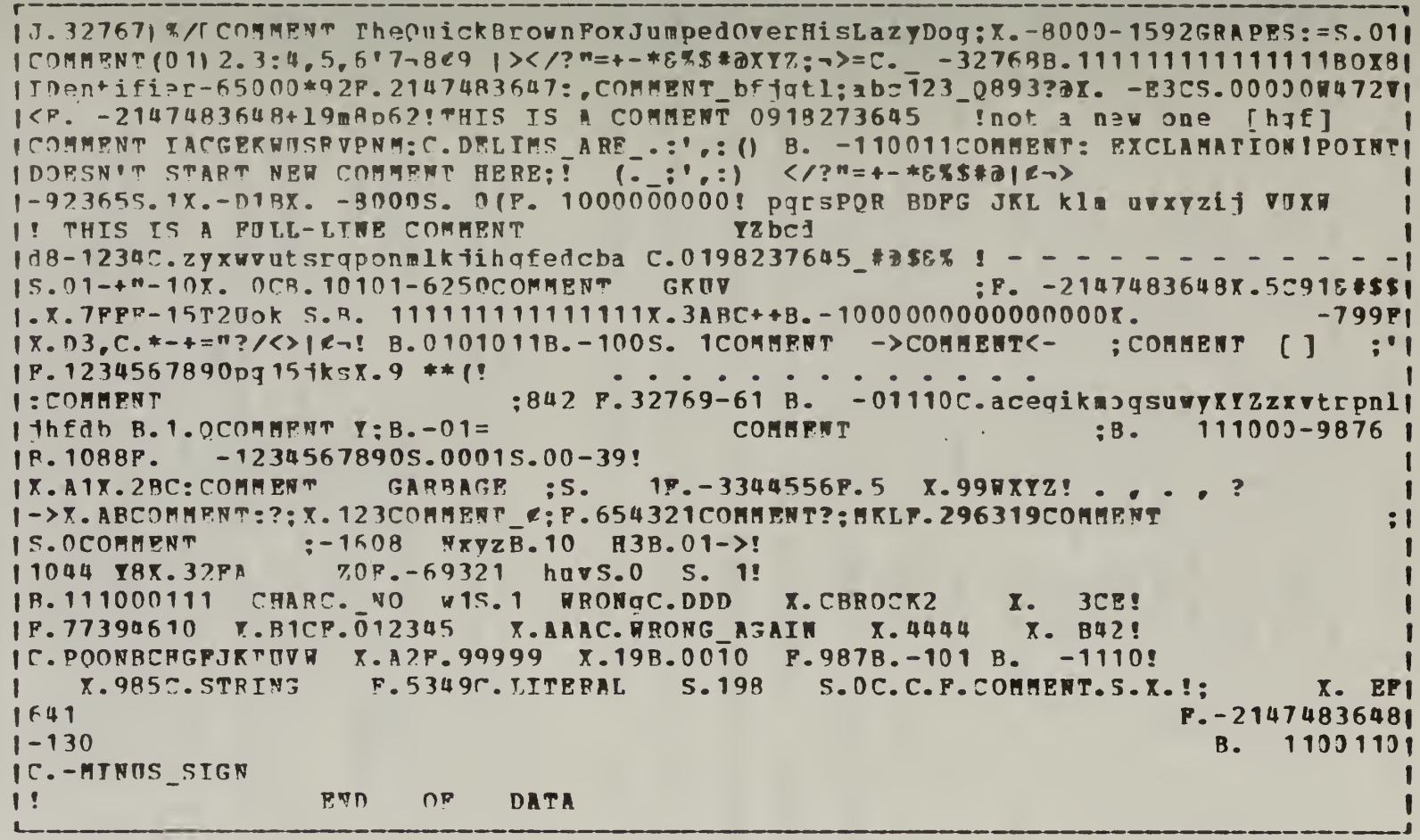

Figure 12

Composite Data for Lexical Analysis Test

4 Syntactic Analysis

The testing of the syntactic analysis phase should encompass the design of test data which would exercise every legal production rule of the CLEOPATRA grammar. Unfortunately, as discussed in Chapter 2, it was virtually impossible to exercise each production rule in every valid context. Therefore, the motivation behind the design of the test data was simply to exercise each legal production at least once; in cases where there were several productions for a single non-terminal (e.g., the many syntactically different ways of writing an ITERATE statement), however, the productions appeared in as many different contexts in the test data as was feasible. In addition, since error-correction facilities are very limited in this implementation, illegal constructs were generally not tested.

Test data for a parser usually consist of syntactically correct but semantic-free program segments. Because syntax and semantics were implemented concurrently in this compiler, the use of semantic-free test programs would have been impractical due to the abundance of semantic error messages that would have been generated. Furthermore, the use of semantically meaningful data in this phase of testing facilitated the tests for semantic correctness, as reported in Chapter 5.

Two main programming examples were chosen as the basis of the remaining testing effort. One is a very skeletal scanner based somewhat on the CLEOPATRA language itself; it is not

intended to be complete, or even correct, but rather is intended to illustrate how one might approach such a task using CLEOPATRA. The second example shows how several user-defined data types might be manipulated. The presentation of these examples is divided into three parts: the first consists of the structure and data blocks, the second incorporates routine blocks (including all statements, but with only the most rudimentary expressions), and the third covers expressions in detail.

4.1 Structure and Data Blocks

This section reports the results obtained from testing four of the types of program blocks in CLEOPATRA: global structure blocks, local structure blocks, global data blocks, and local data blocks. The specification production rules for these blocks are:

```
(4.1)    basic_ref_type ::= INTEGER | LONGINTEGER | BIT |
              CHARACTER
(4.2)    basic_type ::= INTEGER | LONGINTEGER | BIT |
              CHARACTER [(expression)]
(4.4)    ref_type ::= { basic_ref_type | type_name }
              [integer EXTENTS]
(4.5)    type ::= { basic_type | type_name } [array]
(5.1)    structure_block ::= STRUCTURE configuration_name
              { ; link_item }• [;] END configuration_name [;]
```

```
(5.2)    configuration_name ::= procedure_name | type_name |
              operator_link
(5.3)    link_item ::= TYPE type_name [ALIAS identifier] |
              global_link_item
(5.4)    global_link_item ::= PROCEDURE procedure_name
              [ALIAS identifier] [ref_type_list .]
              RETURNS basic_ref_type
(5.5)    global_link_item ::= operator_link : OPERATOR
              [left_ref_types] operator [ALIAS identifier]
              right_ref_types RETURNS basic_type
(5.10)   ref_type_list ::= (type_formal [,type_formal]•)
(5.11)   type_formal ::= ref_type [BY ADDRESS]
(5.12)   left_ref_types ::= ref_type [BY ADDRESS :
              [ref_type_list] | . ref_type_list]
(5.13)   right_ref_types ::= ref_type | : ref_type BY ADDRESS
              | ref_type_list { . ref_type | : ref_type
              BY ADDRESS }
(5.14)   global_structure_block ::= GLOBAL STRUCTURE type_name
              { ; global_link_item }• [;] END type_name [;]
(5.15)   type_name ::= identifier
(5.17)   procedure_name ::= identifier
(5.24)   operator_link ::= identifier
(5.29)   data_block ::= DATA configuration_name
              { ; [CONSTANT | DEFER ] data_group }•
              [;] END configuration_name [;]
(5.30)   global_data_item ::= GLOBAL DATA configuration_name
              { ; [CONSTANT | DEFER] data_group}•
              [;] END configuration_name [;]
```

(5.31) type_data_block ::= GLOBAL DATA type_name
{ ; basic_type [array] item [,item]• }•
[;] END type_name [;]

(5.32) data_group ::= { basic_type | type_name } [array]
item [,item]•

(5.34) array ::= { RIGHT | VECTOR } (bound [,bound]•)

(5.35) bound ::= { *:* | [integer:] integer }

(5.36) item ::= identifier [INIT constant]

These productions are taken from Halbur [9], with the following modifications from Fisher [5]:

1) pointers are not implemented, so they have been deleted from productions 4.1 and 4.2;
2) types and ref_types (productions 4.4 and 4.5) may not have ALIAS names;
3) procedures may return basic_ref_types, not basic_types, as specified by Halbur.

In generating the test data, every form of each production was used at least once; for example, production 5.13 has four forms:

right_ref_type ::= ref_type
: ref_type BY ADDRESS
ref_type_list . ref_type
ref_type_list : ref_type BY ADDRESS

and each one has been exercised. The test data are self-documenting in that internal comments indicate what production rules have been used to generate the code. Figure 13 shows the initial test data.

```
STRUCTURE SYNTAXTEST
        ; PROCEDURE PARSER RETURNS LONGINTEGER
        ; PROCEDURE MANIP ALIAS TYPECHECK RETURNS LONGINTEGER
        ; PROCEDURE OUTPUT(CHARACTER). RETURNS LONGINTEGER
END SYNTAXTEST

STRUCTURE PARSER
        ; TYPE SYMTABENTRY
        ; PROCEDURE SCANNER RETURNS INTEGER
; END PARSER

GLOBAL DATA SYMTABENTRY
        ; INTEGER NAMETABPTR INIT 0,
                  TOKLENGTH INIT 0,
                   TOKTYPE  INIT 0,
                   VALUE    INIT 0
END SYMTABENTRY

GLOBAL STRUCTURE SYMTABENTRY
        ; SETENTRY: OPERATOR #=> (INTEGER,INTEGER,INTEGER).SYMTABENTRY
                RETURNS BIT
END SYMTABENTRY;

GLOBAL DATA PARSER
        ; CONSTANT INTEGER SYMTABMAX INIT 100
        ; SYMTABENTRY RIGHT(100) SYMTAB
        ; CHARACTER(1) VECTOR (1:1000) NAMETAB
        ; BIT EOF INIT S.0
; END PARSER

GLOBAL DATA SCANNER
        ; CONSTANT INTEGER LINELIMIT INIT 80
        ; INTEGER CARDNO INIT 1, CARDPTR INIT 1
        ; CHARACTER(80)CARD, TOKEN INIT C.
        ; CHARACTER(81) TCARD
END SCANNER;

STRUCTURE SCANNER
        ;PROCEDURE LINEIN ALIAS GETLINE(CHARACTER BY ADDRESS).RETURNS BIT
        ;CONSTANTEVAL: OPERATOR |$|(INTEGER).CHARACTER RETURNS LONGINTEGER
        ;PROCEDURE SCANERROR(INTEGER). RETURNS BIT
        ;LOOKUP: OPERATOR LOOKFOR(SYMTABENTRY 1 EXTENTS).CHARACTER
                RETURNS INTEGER
; END SCANNER:

DATA SCANNER
        ;INTEGER I; CHARACTER(1) TCHAR
END SCANNER;

STRUCTURE LINEIN
        ;PROCEDURE INPUT(CHARACTER BY ADDRESS).RETURNS LONGINTEGER
        ;TRANSLATE: OPERATOR @$% ALIAS TRANS(CHARACTER,CHARACTER).
                    CHARACTER RETURNS CHARACTER
END LINEIN;

DATA LINEIN
        ;CHARACTER(80) INPUTSTRING INIT C.
        ;CONSTANT CHARACTER(80) INCHARS INIT C.ABCDEFGHIJKLMNOPQRSTUVWXYZ().:,;'
_@#$%&*-+="?/<>|[¬0123456789!
                                ,TCHARS INIT C.111111111111111111111111112222222
233333333333333333344444444445
END LINEIN

DATA CONSTANTEVAL
        ;INTEGER RADIX INIT 10, SIGN INIT 1, LOW INIT 1,L,DIGIT
        ;LONGINTEGER VALUE INIT F.0
        ;CHARACTER(80) STRING
:END CONSTANTEVAL
```

Figure 13
Structure and Data Block Test Data

```
STRUCTURE MANIP;
       TYPE STACK ALIAS YOYOLIST;
       TYPE MATRIX3BY3;
       TYPE COMPLEX;
       INNERPRODUCT: OPERATOR INTEGER 1 EXTENTS .(INTEGER,INTEGER)
              +* ALIAS DOTPRODUCT (INTEGER).INTEGER 1 EXTENTS
              RETURNS LONGINTEGER;
       PROCEDURE PROCWITH2PARMS(INTEGER,INTEGER). RETURNS BIT;
END MANIP;

DATA MANIP;
       STACK STACK1,STACK2;
       COMPLEX RIGHT(10,5) CR1,CR2;
       MATRIX3BY3 M1,M2
       ; INTEGER INTEXPR
END MANIP;

GLOBAL DATA MANIP;
       INTEGER VECTOR(8) VEC1,VEC2,VEC3
END MANIP;

GLOBAL DATA STACK;
       INTEGER VECTOR(1:100) PDLIST;
       INTEGER TOP INIT 0;
END STACK;

GLOBAL STRUCTURE STACK;
       PUSH: OPERATOR INTEGER PUSH: STACK BY ADDRESS RETURNS BIT;
       POP: OPERATOR INTEGER BY ADDRESS: POP STACK RETURNS BIT;
       INSPECT: OPERATOR STACK.(INTEGER) ??? INTEGER RETURNS INTEGER;
END STACK;

DATA PUSH;
       STACK S1; INTEGER I;
END PUSH;

DATA POP;
       STACK S1; INTEGER I;
END POP;

DATA INSPECT;
       INTEGER Z,I,J; STACK S1

END INSPECT

DATA INNERPRODUCT;
       LONGINTEGER PRODUCT INIT 0.0;
       INTEGER VECTOR (*:*) V1,V2;
       INTEGER I,J,U1,L1,U2
END INNERPRODUCT;

GLOBAL DATA MATRIX3BY3;
       INTEGER VECTOR(3,3) M1,EXTRA;
END MATRIX3BY3;

GLOBAL STRUCTURE MATRIX3BY3;
       MATRIXOP: OPERATOR MATRIX3BY3 BY ADDRESS:(INTEGER) DUALMULT
                 (INTEGER):MATRIX3BY3 BY ADDRESS RETURNS INTEGER
END MATRIX3BY3;

GLOBAL DATA COMPLEX;
       INTEGER INTPART,IMAGPART
END COMPLEX;

GLOBAL STRUCTURE COMPLEX;
       ABS: OPERATOR ABS COMPLEX RETURNS INTEGER;
END COMPLEX;
```

Figure 13 Continued
Structure and Data Block Test Data

The execution of these test data exposed several discrepancies between the language described by the BNF and the language accepted by the parser. These discrepancies and their descriptions are given below.

1) In production 5.1 the semicolon after the "END configuration_name" is not optional, as implied by the BNF. When omitted, an error message is issued and normal processing continues.
2) The operator of production 5.5 cannot be the same as the operator_link. If they are identical, an error message claiming "duplicate declaration" is issued, and the name is no longer recognized as being an operator_link. This causes additional problems later in the program when that operator's definition blocks are encountered.
3) There are several problems with production 5.12. First, the ref_type_list is not optional after "BY ADDRESS:". If it does not appear, every token up to and including the next semicolon is ignored, the token following the semicolon is assumed to be the operator of production 5.5, and processing continues from there as if no error had occurred. Inevitably, more problems arise as a result of the "correction." Second, it appears that the ref_type in production 5.12 cannot include an EXTENTS phrase. When one is included, it causes the next n tokens declared (where n is an unpredictable number, usually on the order of five) all

to be mapped into the same symbol table location as the integer extent number. The most serious side-effect of this is that if the programmer is unfortunate enough to have included an array declaration later in the program, the confusion in the symbol table causes an infinite loop in the array cleanup routine.

4) An interesting deviation involving production 5.36 is that if it appears in any context other than CONSTANT, the identifier is not restricted to being initialized to a constant value; it may be initialized to any previously declared token-name (e.g., a procedure name or a punctuation mark). Only in the CONSTANT context is any type-checking done.

With the exception of the EXTENTS phrase problem in production 5.12 (the cause of which is still unclear), proposals for the repair of these inconsistencies have been documented, but not yet implemented. However, the restrictions imposed by these errors are not overly confining in view of the more serious problems encountered in routine blocks, as described in the next section.

4.2 Routine Blocks

Routine blocks in CLEOPATRA define the actual operations to be carried out by a procedure or operator. The production rules for these blocks (including expressions) are:

(5.16) routine_block ::= PROCEDURE procedure_name
[name_list .] { ; statements }• [;] END
procedure_name [;]

(5.18) name_list ::= (formal [,formal]•)

(5.19) formal ::= identifier

(5.20) procedure_call ::= procedure_name [paramenters]

(5.21) parameters ::= (actual [,actual]•)

(5.22) actual ::= expression

(5.23) routine_block ::= operator_link : OPERATOR [left_names]
operator right_names { ; statement }• [;] END
operator_link [;]

(5.25) left_names ::= ref_type identifier
[{ : | . } name_list | :]

(5.26) right_names ::= [:] ref_type identifier | name_list
{ : | . } ref_type identifier

(5.27) operator_call ::= [expression [{ : | . } [params1]]]
operator [[params2] { : | . }] expression

(5.27a) params1 ::= (actual [,actual]•'

(5.27b) params2 ::= 'actual [,actual]•)

(6.1) expression ::= constant | system_supplied_value |
system_supplied_constant | array_reference |
procedure_call | operator_call | (expression)

(6.2) array_reference ::= array_expression

(6.3) array_expression ::= identifier

(7.1) stmt ::= expression | NIL

(7.2) stmt ::= RETURN expression

```
(7.3)     stmt ::= EXIT [label]

(7.4)     stmt ::= IF expression THEN statement
               [ [;] ELSE statement]

(7.7)     lstmt ::= cstmt | label : cstmt label

(7.8)     label ::= identifier

(7.9)     cstmt ::= BEGIN statement [; statement]• [;] END

(7.10)    cstmt ::= [for_phrase] ITERATE [while_phrase]
               statement [; statement]• [;] [when_phrase] END

(7.11)    for_phrase ::= FOR identifier [ FROM expression]
               STEP expression [[;] ( UPTO | DOWNTO ) expr ] ;

(7.12)    while_phrase ::= WHILE expression ;

(7.13)    when_phrase ::= WHEN expression [;]

(7.14)    cstmt ::= DECISION decision [; decision]•
               [;] ACTION action [; action]•
               [;] [ ELSE statement [;] ] END

(7.15)    decision ::= switch : expression

(7.16)    decision ::= switch {,switch}• [list_layout] :
               list_index

(7.17)    switch ::= identifier

(7.18)    list_layout ::= VECTOR (integer : integer)

(7.19)    list_index ::= expression

(7.20)    action ::= switch_expression : statement

(7.21)    switch_expression ::= [¬] switch [switch_operator [¬]
               switch]•

(7.22)    switch_operator ::= & | | | == | ¬= | AND | OR

(7.23)    statement ::= stmt | lstmt
```

These productions are again based on Halbur [9], but reflect the following modifications introduced by Fisher [5]:

1) parentheses and quotes are used in production 5.27 to disambiguate parameter lists;
2) ALLOCATE and RELEASE statements have been deleted; and
3) a revision of the FOR phrase of the ITERATE statement was made.

The initial test program for these productions is similar to that shown in Figure 14. The routine blocks were incorporated into a revised (i.e., "error"-free) version of this data, and expressions were replaced with single identifiers.

Initial attempts at execution of these test programs were disastrous. A significant number of serious compiler errors were found. The following is a description of these errors:

1) The dot after the name_list in production 5.16 is not accepted at all. If it appears, it is ignored and an error message is issued.
2) Productions 5.25 and 5.26 are implemented as if all occurences of ref_type were replaced with basic_ref_type. This means that the operands of a user-defined operator cannot be arrays or user-defined types, even though these may appear in the original declaration of the operator. If a user-type name or an extent integer does appear as a left_name, the type_name or integer, respectively, is assumed to be the operator, processing continues expecting right_names, and more errors result. If either of

these entities appears in the right_names, an error message is issued, and the remainder of the right_names is ignored with no other apparent side-effect.

3) If at any time during the processing of the statement lists of productions 5.16, 5.23, 7.9 or 7.10 a statement is encountered which begins with any token whose symbol-table number is greater than or equal to 18 and less than 66 (this includes most reserved words and all delimiters, see Halbur [4]), the parser enters an infinite loop. This trap is particularly easy to fall into since the letters B, C, F, S and X are considered reserved words, but no warning is given if they are declared as identifiers.

4) When a compound statement is labelled (via production 7.7) error messages are issued claiming that the label is an undeclared identifier. The identifier is later recognized as a label when the following colon is found, but because of the error messages the program can never be passed on to the code generator and therefore can never be executed.

5) The FOR phrase of production 7.11 is not implemented as Fisher claimed. The form that <u>is</u> implemented is a compromise between the Fisher and Halbur versions:

```
for_phrase ::= FOR identifier [ FROM expression ]
    { [ DOWNTO | UPTO ] expression [;] |
    STEP expression [;] [ [ DOWNTO | UPTO ]
    expression [;] ] }
```

6) A decision statement (production 7.14) cannot be labelled since it is not considered a compound statement in the implementation.

7) Switch expressions (prodution 7.21) do not work for several reasons. First, switch identifiers (production 7.17) are given the type "unknown identifier" in the symbol table instead of being of the type bit as one might expect. Second, the switch operators (production 7.22) & and | are not predefined system operators, so they may not be used at all, and the other switch_operators understandably do not accept unknown identifiers as operands. Even if the switches were of type bit, the wrong type result would be returned.

With the exception of errors 3 and 6, repairs for these discrepencies have been documented but not yet incorporated into the current version of the implementation. Since the FOR phrase as described in 6 is closer to the design goals discussed in Fisher [5] than the specified form, it was decided to leave it as is. Error 3 involves fairly far-reaching side effects, all of which have not yet been tracked down.

4.3 Expressions

It would appear to the casual observer that the setions of this chapter so far have described errors of increasing number and gravity. In this section, unfortunately, the trend is accelerated. To summarize briefly, expressions in this implementation simply do not work.

The specification productions for expressions were listed in section 4.2 and describe expressions whose operators are associated from right to left, along with means for calling on user-defined operators and procedures. For the initial test data attempted, expressions were reincorporated into the routine blocks of Figure 14. The errors uncovered by this experiment are given below.

1) The system-supplied substring operator <- can never be found in the symbol table because of incorrectly initialized buckets and bucket-links used by the lookup routine. More specifically, these links contain two different pointers to the token " (concatenate) but none to <-.

2) Many of the system-supplied operators return the wrong type result. In particular, the comparison operators do not return bit values; instead, they return a result of the type of the two operands being compared. For example, I>J returns an integer result if I and J are of type integer, a character result if I and J are of type character, etc. LBOUND returns a result of type error, and HBOUND, LENGTH, "-> (index) and ?-> (verify) all return integer results. LINT and CHAR return character and longinteger results, respectively, instead of the reverse. Also, several operators do not accept the specified type of operands, notably LENGTH, HBOUND, -> and <-. Figure 15 shows a sample program illustrating these points. (No error messages were issued except in the two places noted in-line!)

```
STRUCTURE SYNTAXTEST
       ; PROCEDURE PARSER RETURNS LONGINTEGER
       ; PROCEDURE MANIP ALIAS TYPECHECK RETURNS LONGINTEGER
       ; PROCEDURE OUTPUT(CHARACTER). RETURNS LONGINTEGER
END SYNTAXTEST

STRUCTURE PARSER
       ; TYPE SYMTABENTRY
       ; PROCEDURE SCANNER RETURNS INTEGER
; END PARSER

GLOBAL DATA SYMTABENTRY
       ; INTEGER NAMETABPTR INIT 0,
                 TOKLENGTH  INIT 0,
                 TOKTYPE    INIT 0,
                 VALUE      INIT 0
END SYMTABENTRY

GLOBAL STRUCTURE SYMTABENTRY
       ; SETENTRY: OPERATOR #=> (INTEGER,INTEGER,INTEGER).SYMTABENTRY
             RETURNS BIT
END SYMTABENTRY;

GLOBAL DATA PARSER
       ; CONSTANT INTEGER SYMTABMAX INIT 100
       ; SYMTABENTRY RIGHT(100) SYMTAB
       ; CHARACTER(1) VECTOR (1:1000) NAMETAB
       ; BIT EOF INIT S.0
; END PARSER

PROCEDURE PARSER
       ; ITERATE SCANNER WHEN EOF == S.1 END
END PARSER

GLOBAL DATA SCANNER
       ; CONSTANT INTEGER LINELIMIT INIT 80
       ; INTEGER CARDNO INIT 1, CARDPTR INIT 1
       ; CHARACTER(80)CARD, TOKEN INIT C.
       ; CHARACTER(81) TCARD
END SCANNER;

STRUCTURE SCANNER
       ;PROCEDURE LINEIN ALIAS GETLINE(CHARACTER BY ADDRESS).RETURNS BIT
       ;CONSTANTVAL: OPERATOR |#|(INTEGER).CHARACTER RETURNS LONGINTEGER
       ;PROCEDURE SCANERROR(INTEGER). RETURNS BIT
       ;LOOKUP: OPERATOR LOOKFOR(SYMTABENTRY 1 EXTENTS).CHARACTER
             RETURNS INTEGER
; END SCANNER;

DATA SCANNER
       ;INTEGER I; CHARACTER(1) TCHAR
END SCANNER;

PROCEDURE SCANNER
       ; GETTOKEN: ITERATE WHILE TRUE;
       DECISION LETTER,DELIMITER,SPECIAL,DIGIT,CONTROL,EOSR :
              CHAR 1 -> TCARD <- CARDPTR;
       ACTION;
       LETTER: BEGIN FOR I FROM CARDPTR+1 UPTO LINELIMIT;
                     ITERATE TOKEN:=TOKEN"1->CARD<-I - 1;
                           TCHAR := 1->TCARD<-I;
                     WHEN (TCHAR¬=C.1 ) AND TCHAR¬=C.4 END;
                     RETURN LOOKFOR'SYMTAB).TOKEN
              END;
       LETTER AND DIGIT: NIL; !IMPOSSIBLE CASE
       DELIMITER: TOKEN:=1->CARD<-I ;
       CONTROL OR EOSR: IF LINEIN(CARD) THEN SCANERROR(1) ELSE EXIT;
       DIGIT: TCHAR := C.4 ;
       SPECIAL: TCHAR := C.3 ;
```

Figure 14
Routine Block Test Data

```
        DIGIT OR SPECIAL : BEGIN TOKEN :=
                        (CARDPTR:=TCHAR ?-> TCARD <-
                        CARDPTR+1) -> CARD ) <- CARDPTR;
                        RETURN LOOKFOR'SYMTAB).TOKEN;
                        END
        ELSE SCANERROR(2) !UNDEFINED CHARACTER
        END !DECISION
        END GETTOKEN
END SCANNER

STRUCTURE LINEIN
        ;PROCEDURE INPUT(CHARACTER BY ADDRESS).RETURNS LONGINTEGER
        ;TRANSLATE: OPERATOR @$% ALIAS TRANS(CHARACTER,CHARACTER).
              CHARACTER RETURNS CHARACTER
END LINEIN;

DATA LINEIN
        ;CHARACTER(80) INPUTSTRING INIT C.
        ;CONSTANT CHARACTER(80) INCHARS INIT C.ABCDEFGHIJKLMNOPQRSTUVWXYZ().:,;'
_@$%&*-+="?/<>|[¬0123456789!
                                ,TCHARS INIT C.11111111111111111111111112222222
233333333333333333344444444445
END LINEIN

PROCEDURE LINEIN(INPUTSTRING).;
        IF INPUT(INPUTSTRING)==F.0
        THEN RETURN FALSE;
        ELSE BEGIN CARDNO := CARDNO+1;
                   CARDPTR := 1;
                   OUTPUT((CHAR CARDNO)"C.|_ "INPUTSTRING"C._| ;
                   TCARD := TRANS'INCHARS,TCHARS).CARD"C.6 ;
                   RETURN TRUE
                END;
END LINEIN;

DATA CONSTANTEVAL
        ;INTEGER RADIX INIT 10, SIGN INIT 1, LOW INIT 1,L,DIGIT
        ;LONGINTEGER VALUE INIT F.0
        ;CHARACTER(80) STRING
;END CONSTANTEVAL

CONSTANTEVAL:OPERATOR |*|(RADIX).CHARACTER STRING
        ; IF C.- == (I:=1)->STRING
        THEN BEGIN SIGN := -1; I:=2 END;
        FOR I UPTO LENGTH STRING ITERATE
               DIGIT := ( 1 -> STRING <- I)"-> C.0123456789ABCDEF ;
               IF DIGIT ¬= INTEGER NIL
               THEN VALUE := (DIGIT - 1)+RADIX*VALUE
               ELSE RETURN F.0
        END;
        RETURN SIGN*VALUE
END CONSTANTEVAL

STRUCTURE MANIP;
        TYPE STACK ALIAS YOYOLIST;
        TYPE MATRIX3BY3;
        TYPE COMPLEX;
        INNERPRODUCT: OPERATOR INTEGER 1 EXTENTS .(INTEGER,INTEGER)
               ** ALIAS DOTPRODUCT (INTEGER).INTEGER 1 EXTENTS
               RETURNS LONGINTEGER;
        PROCEDURE PROCWITH2PARMS(INTEGER,INTEGER). RETURNS BIT;
END MANIP;

DATA MANIP;
        STACK STACK1,STACK2;
        COMPLEX RIGHT(10,5) CR1,CR2;
        MATRIX3BY3 M1,M2
        ; INTEGER INTEXPR
END MANIP;
```

Figure 14 Continued
Routine Block Test Data

```
GLOBAL DATA MANIP;
       INTEGER VECTOR(8) VEC1,VEC2,VEC3
END MANIP;

PROCEDURE MANIP;
       NONSENSE: BEGIN
       PROCWITH2PARMS(S.0,B.01001);
       VEC1.(4,8 ' ** ' 2).VEC2;
       DECISION SW4,SW5,SW6 VECTOR(4:6) : INTEXPR;
              SW1,SW2 : INTEXPR;
              SW0 : FALSE
       ACTION;
              ¬SW0 == SW4 : EXIT NONSENSE;
              SW5 ¬= SW2 AND SW1 : RETURN F.214846
              ELSE RETURN F.0
       END
       END NONSENSE
END MANIP

GLOBAL DATA STACK;
       INTEGER VECTOR(1:100) PDLIST;
       INTEGER TOP INIT 0;
END STACK;

GLOBAL STRUCTURE STACK;
       PUSH: OPERATOR INTEGER PUSH: STACK BY ADDRESS RETURNS BIT;
       POP: OPERATOR INTEGER BY ADDRESS: POP STACK RETURNS BIT;

       INSPECT: OPERATOR STACK.(INTEGER) ??? INTEGER RETURNS INTEGER;
END STACK;

DATA PUSH;
       STACK S1; INTEGER I;
END PUSH;

PUSH: OPERATOR INTEGER I PUSHI : STACK S1 ;
       DECISION OVERFLOW : 100 <= TOP.S1
       ACTION; ¬OVERFLOW : BEGIN
                           PDLIST.S1(TOP.S1:=TOP.S1+1):=I;
                           RETURN TRUE;
                           END;
              OVERFLOW : RETURN FALSE ;
       END
END PUSH

DATA POP;
       STACK S1; INTEGER I;
END POP;

POP: OPERATOR INTEGER I : POPI STACK S1;
       DECISION UNDERFLOW: TOP.S1 <= 1
       ACTION; UNDERFLOW: RETURN FALSE;
              ELSE BEGIN
                     I := PDLIST.S1(TOP.S1);
                     TOP.S1 := TOP.S1- 1;
                     RETURN TRUE;
                     END;
END POP;

DATA INSPECT;
       INTEGER Z,I,J; STACK S1
END INSPECT

INSPECT: OPERATOR STACK S1.(I) ??? INTEGER X;
       FOR J FROM TOP.S1 STEP -1 ITERATE WHILE J<=I;
       IF X == PDLIST.S1(J) THEN RETURN J
       END;
       RETURN 0
END INSPECT
```

Figure 14 Continued
Routine Block Test Data

```
DATA INNERPRODUCT;
       LONGINTEGER PRODUCT INIT F.0;
       INTEGER VECTOR (*:*) V1,V2;
       INTEGER I,J,U1,L1,U2
END INNERPRODUCT;

INNERPRODUCT: OPERATOR INTEGER 1 EXTENTS V1.(U1,L1) **
              (U2).INTEGER 1 EXTENTS V2;
       J:=1 + 1 HBOUND V2;
       FOR I FROM 1 HBOUND V1 DOWNTO 1 LBOUND V1
       ITERATE PRODUCT := PRODUCT+V1(I)*V2(J:=J- 1)
       END;
       RETURN PRODUCT
END INNERPRODUCT;

GLOBAL DATA MATRIX3BY3;
       INTEGER VECTOR(3,3) M1,EXTRA;
END MATRIX3BY3;

GLOBAL STRUCTURE MATRIX3BY3;
       MATRIXOP: OPERATOR MATRIX3BY3 BY ADDRESS:(INTEGER) DUALMULT
              (INTEGER):MATRIX3BY3 BY ADDRESS RETURNS INTEGER
END MATRIX3BY3;

GLOBAL DATA COMPLEX;
       INTEGER INTPART,IMAGPART
END COMPLEX;

GLOBAL STRUCTURE COMPLEX;
       ABS: OPERATOR ABS COMPLEX RETURNS INTEGER;
END COMPLEX;
```

Figure 14 Continued
Routine Block Test Data

```
STRUCTURE EXPR
        ; PROCEDURE OUTPUT(CHARACTER).RETURNS LONGINTEGER
END EXPR;

DATA EXPR
        ; INTEGER I,J,K
        ; LONGINTEGER W,Y,Z
        ; BIT B1,B2,B3
        ; INTEGER VECTOR(10,100) V1,V2
        ; CHARACTER(5) C1,C2,C3
END EXPR;

PROCEDURE EXPR;
        J := - I + ABS J ;              ! 84,66,68,67
        K := I * J // K + 1 ;           ! 69,70,71,84
        K := ( I := I - 1 ) * J := J // 2 ;
        B1 := B1 AND B2 OR ¬B3;         ! 74,76,120,77
        K := (J:=I MOD 5)**J;           ! 73,72,84
        I := 2 LBOUND V1 ;    ! 75, SHOULD GET ERROR MESSAGE ON :=
                              ! SINCE LBOUND RETURNS "ERROR" TYPE
        I := J==3;      ! 78
        I := J¬=3;      ! 79
        I := J>3;       ! 80
        I := J<3;       ! 81
        I := J>=3;      ! 82
        I := J<=3;      ! 83
        C2 := LENGTH C1;        ! 86
        C1 HBOUND C1 ;          ! 87
        C3 := C1 -> C2;         ! 88
        C3 := C1 <- C2;         ! 89 SHOULD GET ERROR MESSAGE ON <-
        C3 := C1 ¬-> C2;        ! 90
        C3 := C1 ?-> C2;        ! 91
        C2 := LINT C1;          ! 93,119
        W := CHAR W;            ! 94,121
        Y := ABS - W;           ! 95,96
        W := W+Y*Z//P.2;        ! 97,98,99,100
        Z := W**Y;              ! 101
        Z := W MOD Y;           ! 102
        W := Y==P.10;           ! 103
        B1 := B2==B3;           ! 104
        C1 := C2 ;              ! 105
        W := Y¬=Z;              ! 106
        B1 := B2 ¬= B3;         ! 108
        C1 := C2 ¬= C3;         ! 109
        W := Y>Z;               ! 111
        C1 := C2>C3;            ! 112
        W := Y<Z;               ! 113
        C1 := C2<C3;            ! 114
        W := Y>=Z;              ! 115
        C1 := C2>=C3;           ! 116
        W:=Y<=Z;                ! 117
        C1:=C2<=C3;             ! 118
        C3 := C1 ¬ C1 ;         ! 92
END EXPR;
```

Figure 15
Unusual Expressions Accepted

3) Productions 5.27 and 5.27b are not accepted for unary operators. For example,

LOOKFOR'SYMTAB).TOKEN

is not accepted on the grounds of mismatched parentheses. On the assumption that the specifications were not complete (i.e., unary operators were perhaps considered a special case), the following was tried:

LOOKFOR('SYMTAB).TOKEN .

This caused the parser to enter a loop that vaccilated between the quote and some other unspecified token, printing repeatedly

Illegal or inactive token. Bad token was '

Illegal or inactive token. Bad token was

Illegal or inactive token. Bad token was '

until a message was issued claiming the statement was too long. This operator call was tried one last time deleting the quote altogether, but the parser objected to the dot.

There are many more errors in the processing of expressions, too numerous to list and too nebulous to describe intelligently. Unfortunately, most of these cause fatal IBM/360 errors which made further testing fruitless.

5 Semantic Analysis

As stated before, it was difficult to separate pure syntax from semantics since these were implemented concurrently. Section 4 covered type-checking and some other semantics along with syntax. This section is concerned with the correctness of the symbol table complex and the intermediate text.

5.1 Symbol Table Complex

The major data base for the compiler is the symbol table and includes the name table, the hash table, the type analysis table (token attributes and some scope information), the constant table, the level/configuration table (activation information), the configuration table, and the type table (parameter information). These tables are first inspected for the correctness of the system predefined symbol entries and then for user-defined entries.

5.1.1 Predefined Symbols

The system's predefined symbols are locations 1 through 122, and all of these entries in all the tables listed above are initialized via the PL/I INIT option before the compilation of the user program.

The hash table is incorrectly initialized as discussed in section 4.3: the entry <- has no pointer to it, while " has two pointers, one of which is never used. Unfortunately, the repair

is not as simple as changing the unused "-pointer to point to <-; the repair is documented elsewhere. All other entries are correct.

The type field (giving the type returned or the type of the identifier) of the type analysis table is also incorrectly initialized, as discussed in section 4.3. Entries 78 through 83 and 103 through 118, the comparison operators, should contain 8 indicating a bit value is to be returned. The returned types of LINT and CHAR are correctly initialized to longinteger and character respectively, but in actuality they return the reverse of this. (It is unclear why this is the case.) LENGTH, HBOUND, "-> and ?-> are initialized to return character results, but all of these should return integer results. All other entries of the type analysis table are correct.

The type table field defining the types of procedure parameters and operator operands also has some incorrectly initialized entries. The operators HBOUND, -> and <- are all set to accept character operands while the first should take an integer operand and the last two should take an integer and a character operand and the reverse, respectively. Again the entries of LINT and CHAR appear to be reversed. All other entries are correct.

The name table, the constant table, the configuration tble and the level/configuration table are correctly initialized for the predefined symbols.

5.1.2 User-defined Symbols

Information about user-defined symbols is added to the symbol table as it is determined from the syntax. For symbols which occur in any error context, the information in its entry may not be complete or correct, which is understandable. However, in some cases these errors can have adverse side-effects on other entries in the symbol table by overwriting them.

For instance, the token :=> was mistakenly entered as an operator. In the operator definition in the program, the symbol table lookup routine was in declare-token mode when it reached the :=>, correctly isolated it as the special symbol :=, and proceeded to change that entry to be a unary operator, which :=> was supposed to be. Then when the parser reached the >, it correctly issued an error message claiming an unexpected token, but entered the return type of error in the := symbol table location (and the one before it was changed also, since configurations use two consecutive rows of the symbol table). The reason this is mentioned is that if later in the program there are integer assignment statements, much confusion will result. The point is that the predefined symbol entries should have a higher degree of protection.

With the exception of the problem described in section 4.1 which caused several different tokens to be placed in the same location of the symbol table, thus making more than one hash link point to that location, no other errors have been detected in the hash links of inserted items.

No errors have been detected in the configuration table and level/configuration table, with one exception: all the predecessor links in the level/configuration table remain zero throughout the compilation of any program. Perhaps this pointer is not explicitly used, since the "father" link of the configuration table seems to contain the same information. Also no errors have been found in the manipulation of the constant table, name table or the type table except for those tokens found in an error context.

5.2 Intermediate Text

One problem with the testing of the intermediate text was that it was difficult to construct a program that had few errors, therefore a readable intermediate text. A second problem was that by this point in the project it had become clear that the CLEOPATRA code generator contained some serious incompatibilities which prevented the linking of the two passes of the compiler. But the programs presented thus far were "doctored up" a bit and the intermediate text from their compilation was analyzed. Happily enough, no errors were detected.

It might seem somewhat suspicious that the semantics of nesting levels and intermediate text generation, much more complicated tasks than lexical and syntactic analysis, would be relatively error-free, while the "easier" parts of the compiler were bug-ridden. There could be two reasons for this. Perhaps the implementor spent a greater amount of time and care in constructing the more difficult parts, knowing that the chance of

error was greater, and spent less time and attention to detail on the "easier" parts. Also it could be true that the test programs that could be used for testing in this phase were very simple, due to the errors in the previous phase of the compiler. Be that as it may, Chapter 4 certainly documents enough errors for two chapters.

6 Conclusions

The three parts of the CLEOPATRA analysis phase have now been tested. A thorough test of the lexical analysis uncovered a moderate number of errors, all of which were rectified, thus establishing a high level of reliability for the scanner. Reasonably thorough tests, considering the complexity of CLEOPATRA, were run against the syntax analyser; again only a moderate number of purely syntactic discrepancies were found (but not yet corrected). When the documented repairs are implemented, the parser will be acceptably close to the specifications. However, the syntax tests were confounded by the shambles of the type-checking and expression-handling portions of the semantic analyzer. The remainder of the semantic analyzer, intermediate text generation, is fairly good.

Evaluation of this project is difficult. In one sense, it was very successful since so many errors were found, and that was the purpose of the testing. However, one of the basic assumptions of a validation effort is that the code has been debugged already by the implementor to remove the more mechanical errors of coding; it is difficult to believe that this was done. Too many of the errors found would have become immediately obvious if even the most superficial testing had been performed. For example, any test program that contained even a very simple expression (and one that does not is rare) would have caused the compiler to come crashing down. As a result, this project was "successful" as a <u>debugging</u> venture.

Unfortunately, the project was not as successful as a validation effort; the answer to the question "Is this implementation of a CLEOPATRA analysis phase a valid one?" would be an unequivocal "NO!". At least not for this version of the implementation, because the validation process is not really completed. The documented errors must first be repaired, then the tests must be rerun. No doubt more errors will be found requiring the process to be repeated.

Despite the pessimism exhibited above, the future of the CLEOPATRA project is promising. Plans include the production an an improved version of the implementation to incorporate corrections of the errors described in this thesis, and the retesting of the new version using the data that has been presented here. The result should be a well-written, reliable analysis phase implementation which will provide a solid basis for future CLEOPATRA expansion.

References

[1] Schreiner, Axel T., "CLEOPATRA Comprehensive Language for Elegant Operating System and Translator Design", Report UIUCDCS-R-74-646, Department of Computer Science, University of Illinois, May 1974.

[2] ---------, "CLEOPATRA A Proposal for Another System Implementation Language", Report UIUCDCS-R-74-654, Department of Computer Science, University of Illinois, June 1974.

[3] Halbur, John D., "A Code Generator for the CLEOPATRA Language", Report UIUCDCS-R-75-739, Department of Computer Science, University of Illinois, July 1975.

[4] ------, "CLEOPATRA Code Generator User's Guide", Report UIUCDCS-R-76-740, Department of Computer Science, University of Illinois, January 1976.

[5] Fisher, Scott H., "Implementation of the Language CLEOPATRA: the Analysis Pass," Master's thesis, University of Illinois, September 1976.

[6] Gruenberger, Fred, "Program Testing and Validating", Datamation, Vol. 14, No. 7 (July 1968), pp. 39-47.

[7] Elmendorf, W. R., "Controlling the Functional Testing of an Operating System", IEEE Transactions on Systems Science and Cybernetics, Vol. SSC-5, No. 4 (October 1969), pp. 284-290.

[8] Brown, A. R., and Sampson, W. A., Program Debugging, American Elsevier, New York, 1973.

[9] Goodenough, John B., and Gerhart, Susan L., "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2 (June 1975), pp. 156-173.

[10] London, R. L., "Proving Program Correctness: Some Techniques and Examples," BIT, Vol. 10, No. 2 (September 1970), pp. 168-182.

[11] Huang, J. C., "An Approach to Program Testing", Computing Surveys, Vol. 7, No. 3 (September 1975), pp. 113-128.

[12] Hetzel, William C., "Principles of Computer Program Testing", Program Test Methods, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

Appendix: Summary of Productions Implemented

This appendix summarizes the productions that define CLEOPATRA as it is implemented. Upper-case symbols and punctuation designate terminal symbols of the language; lower-case symbols represent non-terminals. Other notation used is:

|	indicates a choice
{ }	combine a number of choices
[ ]	the enclosed entity may appear or may be omitted
[ ]•	the enclosed entity may appear zero or more times
{ }•	the enclosed entity must appear one or more times

Since the character | is also a member of the CLEOPATRA character set, it will be underlined (|) when it denotes itself as a terminal symbol.

(1.1) letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

(1.2) delimiting_character ::= (|) | . | : | , | ' | ; | blank_character | _

(1.3) special_character ::= @ | # | $ | % | & | * | - | + | = | " | ? | / | < | > | ⊥ | ≠ | ¬

(1.4) digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(1.5) control ::= !

(1.6) comment ::= COMMENT any sequence of characters with the exception of a semicolon ;

(1.7) ::= ! any sequence of characters with the exception of an end of the source record end_of_source_record

(2.1) identifier ::= letter [letter | digit]•

(2.2) operator ::= {special_character}• | identifier

(2.3) constant ::= integer | long_integer | bit |
literal_value

(2.4) decimal_digit ::= digit

(2.5) hexadecimal_digit ::= digit | A | B | C | D | E | F

(2.7) binary_digit ::= 0 | 1

(2.8) minus_symbol ::= -

(2.9) decimal_string ::= [minus_symbol] {decimal_digit}•

(2.10) hexidecimal_string ::= X. [minus_symbol]
{hexadecimal_digit}•

(2.12) binary_string ::= B. [minus_symbol] {binary_digit}•

(2.13) basic_constant ::= decimal_string | hexadecimal_string
| binary_string

(2.14) integer ::= basic_constant

(2.15) long_integer ::= F. decimal_string

(2.17) bit ::= S. binary_digit

(2.28) literal_value ::= C. any sequence of characters up to
but not including the first following
blank_character

(2.29) system_supplied_value ::= basic_type NIL |
{ LONGINTEGER | INTEGER } { LARGE | SMALL }

(2.30) system_supplied_constant ::= FALSE | FIRST | LAST |
TRUE

(4.1) basic_ref_type ::= INTEGER | LONGINTEGER | BIT |
CHARACTER

(4.2) basic_type ::= INTEGER | LONGINTEGER | BIT |
CHARACTER [(expression)]

(4.4) ref_type ::= { basic_ref_type | type_name }
[integer EXTENTS]

```
(5.1)    structure_block ::= STRUCTURE configuration_name
              { ; link_item }• [;] END configuration_name ;
(5.2)    configuration_name ::= procedure_name | type_name |
              operator_link
(5.3)    link_item ::= TYPE type_name [ALIAS identifier] |
              global_link_item
(5.4)    global_link_item ::= PROCEDURE procedure_name
              [ALIAS identifier] [ref_type_list .]
              RETURNS basic_ref_type
(5.5)    global_link_item ::= operator_link : OPERATOR
              [left_ref_types] operator [ALIAS identifier]
              right_ref_types RETURNS basic_type
(5.10)   ref_type_list ::= (type_formal [,type_formal]•)
(5.11)   type_formal ::= ref_type [BY ADDRESS]
(5.12)   left_ref_types ::= { basic_ref_type | type_name }
              [BY ADDRESS :
              ref_type_list |
              . ref_type_list]
(5.13)   right_ref_types ::= ref_type | : ref_type BY ADDRESS
              | ref_type_list { . ref_type | : ref_type
              BY ADDRESS }
(5.14)   global_structure_block ::= GLOBAL STRUCTURE type_name
              { ; global_link_item }• [;] END type_name [;]
(5.15)   type_name ::= identifier
(5.16)   routine_block ::= PROCEDURE procedure_name
              [name_list] { ; statements }• [;] END
              procedure_name [;]
```

```
(5.17)    procedure_name ::= identifier

(5.18)    name_list ::= (formal [,formal]•)

(5.19)    formal ::= identifier

(5.20)    procedure_call ::= procedure_name [paramenters]

(5.21)    parameters ::= (actual [,actual]•)

(5.22)    actual ::= expression

(5.23)    routine_block ::= operator_link : OPERATOR [left_names]
               operator right_names { ; statement }• [;] END
               operator_link [;]

(5.24)    operator_link ::= identifier

(5.25)    left_names ::= basic_ref_type identifier
               [ { : | . } name_list | : ]

(5.26)    right_names ::= [:] basic_ref_type identifier |
               name_list
               { : | . } basic_ref_type identifier

(5.27)    operator_call ::= [expression [ { : | . } [params1]]]
               operator [ [params2] { : | . }] expression

(5.27a)   params1 ::= (actual [,actual]•'

(5.27b)   params2 ::= 'actual [,actual]•)

(5.29)    data_block ::= DATA configuration_name
               { ; [CONSTANT | DEFER ] data_group }•
               [;] END configuration_name [;]

(5.30)    global_data_item ::= GLOBAL DATA configuration_name
               { ; [CONSTANT | DEFER] data_group}•
               [;] END configuration_name [;]

(5.31)    type_data_block ::= GLOBAL DATA type_name
               { ; basic_type [array] item [,item]• }•
               [;] END type_name [;]
```

(5.32) data_group ::= { basic_type | type_name } [array]
item [,item]•

(5.34) array ::= { RIGHT | VECTOR } (bound [,bound]•)

(5.35) bound ::= { *:* | [integer:] integer }

(5.36) item ::= identifier [INIT constant]

(6.1) expression ::= constant | system_supplied_value |
system_supplied_constant | array_reference |
procedure_call | operator_call | (expression)

(6.2) array_reference ::= array_expression

(6.3) array_expression ::= identifier

```
(7.1)     stmt ::= expression | NIL

(7.2)     stmt ::= RETURN expression

(7.3)     stmt ::= EXIT [label]

(7.4)     stmt ::= IF expression THEN statement
               [ [;] ELSE statement ]

(7.7)     lstmt ::= cstmt | label : cstmt label

(7.8)     label ::= identifier

(7.9)     cstmt ::= BEGIN statement [; statement]• [;] END

(7.10)    cstmt ::= [for_phrase] ITERATE [while_phrase]
               statement [; statement]• [;] [when_phrase] END

(7.11)    for_phrase ::= FOR identifier [ FROM expression ]
               { [ DOWNTO | UPTO } expression [;] |
               STEP expression [;] [ { DOWNTO | UPTO }
               expression [;] ] }

(7.12)    while_phrase ::= WHILE expression ;

(7.13)    when_phrase ::= WHEN expression [;]

(7.14)    stmt ::= DECISION decision [; decision]•
               [;] ACTION action [; action]•
               [;] [ ELSE statement [;] ] END

(7.15)    decision ::= switch : expression

(7.16)    decision ::= switch{,switch}• [list_layout] :
               list_index

(7.17)    switch ::= identifier

(7.18)    list_layout ::= VECTOR (integer : integer)

(7.19)    list_index ::= expression

(7.20)    action ::= switch_expression : statement
```

(7.21) switch_expression ::= [¬] switch [switch_operator [¬] switch]•

(7.22) switch_operator ::= == | ¬= | AND | OR

(7.23) statement ::= stmt | lstmt

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-76-828	2.	3. Recipient's Accession No.

Title and Subtitle

Validation of the Analysis Pass of the CLEOPATRA Compiler

5. Report Date: October 1976

6.

Author(s)

Sandra Ann Leach

8. Performing Organization Rept. No.

Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

Sponsoring Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

13. Type of Report & Period Covered: Master's Thesis

14.

Supplementary Notes

Abstracts

An implementation of the analysis pass of the compiler for a subset of the CLEOPATRA language has been recently produced. This report describes an attempt at the validation of this software performed concurrently with the implementation. Covered in detail are the design, production and execution of test cases for lexical, syntactic and semantic phases. The results of this work are the thorough documentation of the discrepancies between the subset specifications and the implementation and the generation of a set of test data to be permanently associated with the CLEOPATRA compiler.

Key Words and Document Analysis. 17a. Descriptors

Programming languages
Compilation
Compiling
Verification
Validation

Identifiers/Open-Ended Terms

CLEOPATRA

COSATI Field/Group

Availability Statement	19. Security Class (This Report)	21. No. of Pages
UNLIMITED	UNCLASSIFIED	72
	20. Security Class (This Page) UNCLASSIFIED	22. Price

JAN 2 5 1977

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.824-829(1978
Internal report /
3 0112 088402992